加微信：642945106 发送“赠送”领取赠送精品课程
发数字“2”获取众筹列表

13 | ControllerEventManager：变身单线程后的Controller如何处理事件?

2020-05-19 胡夕

讲述：胡夕
时长 18:01 大小 16.51M

你好，我是胡夕。 今天，我们来学习下 Controller 的单线程事件处理器源码。

所谓的单线程事件处理器，就是 Controller 端定义的一个组件。该组件内置了一个专属线程，负责处理其他线程发送过来的 Controller 事件。另外，它还定义了一些管理方法，用于为专属线程输送待处理事件。

在 0.11.0.0 版本之前，Controller 组件的源码非常复杂。集群元数据信息在程序中同时被多个线程访问，因此，源码里有大量的 Monitor 锁、Lock 锁或其他线程安全机制，这就导致，这部分代码读起来晦涩难懂，改动起来也困难重重，因为你根本不知道，变动了这个线

程访问的数据，会不会影响到其他线程。同时，开发人员在修复 Controller Bug 时，也非常吃力。

鉴于这个原因，自 0.11.0.0 版本开始，社区陆续对 Controller 代码结构进行了改造。其中非常重要的一环，就是将**多线程并发访问的方式改为了单线程的事件队列方式**。

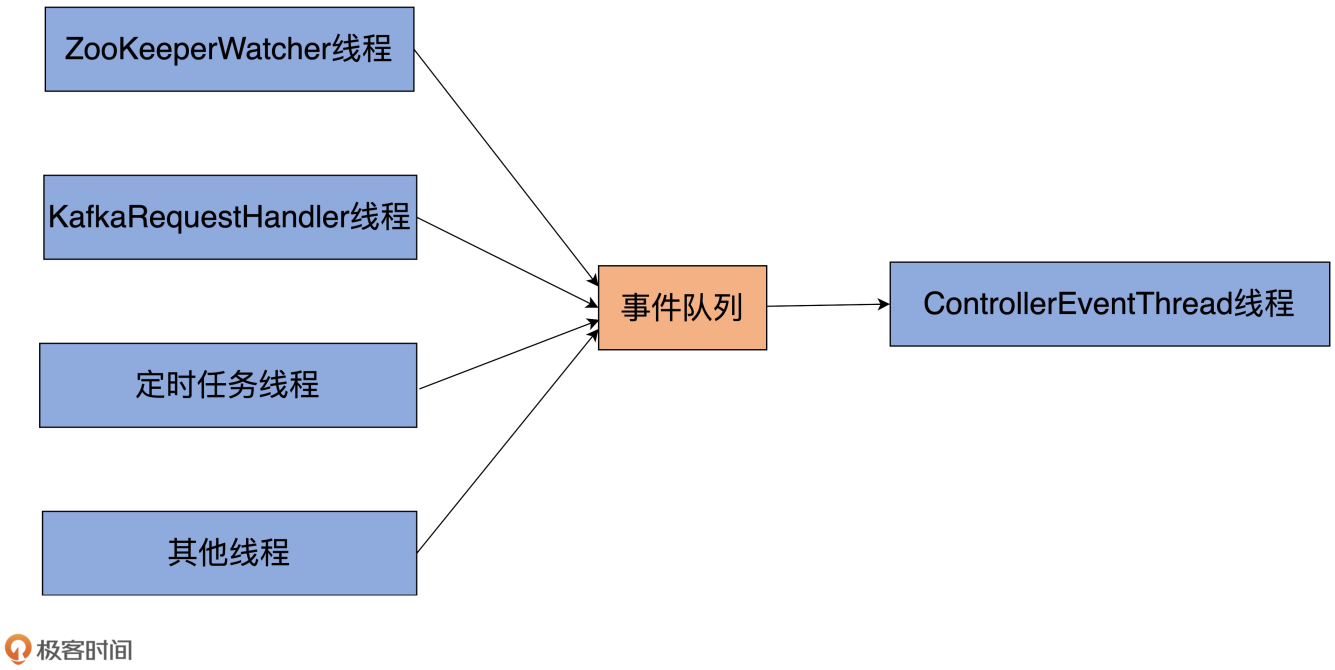
这里的单线程，并非是指 Controller 只有一个线程了，而是指**对局部状态的访问限制在一个专属线程上**，即让这个特定线程排他性地操作 Controller 元数据信息。

这样一来，整个组件代码就不必担心多线程访问引发的各种线程安全问题了，源码也可以抛弃各种不必要的锁机制，最终大大简化了 Controller 端的代码结构。

这部分源码非常重要，**它能够帮助你掌握 Controller 端处理各类事件的原理，这将极大地提升你在实际场景中处理 Controller 各类问题的能力**。因此，我建议你多读几遍，彻底了解 Controller 是怎么处理各种事件的。

基本术语和概念

接下来，我们先宏观领略一下 Controller 单线程事件队列处理模型及其基础组件。



从图中可见，Controller 端有多个线程向事件队列写入不同种类的事件，比如，ZooKeeper 端注册的 Watcher 线程、KafkaRequestHandler 线程、Kafka 定时任务线

程，等等。而在事件队列的另一端，只有一个名为 ControllerEventThread 的线程专门负责“消费”或处理队列中的事件。这就是所谓的单线程事件队列模型。

参与实现这个模型的源码类有 4 个。

ControllerEventProcessor：Controller 端的事件处理器接口。

ControllerEvent：Controller 事件，也就是事件队列中被处理的对象。

ControllerEventManager：事件处理器，用于创建和管理 ControllerEventThread。

ControllerEventThread：专属的事件处理线程，唯一的作用是处理不同种类的 ControllEvent。这个类是 ControllerEventManager 类内部定义的线程类。

今天，我们的重要目标就是要搞懂这 4 个类。就像我前面说的，它们完整地构建出了单线程事件队列模型。下面我们将一个一个地学习它们的源码，你要重点掌握事件队列的实现以及专属线程是如何访问事件队列的。

ControllerEventProcessor

防止断更 请务必加首发微信：1716143665

这个接口位于 controller 包下的 ControllerEventManager.scala 文件中。它定义了一个支持普通处理和抢占处理 Controller 事件的接口，代码如下所示：

复制代码

```
trait ControllerEventProcessor {
  def process(event: ControllerEvent): Unit
  def preempt(event: ControllerEvent): Unit
}
```

该接口定义了两个方法，分别是 process 和 preempt。

process：接收一个 Controller 事件，并进行处理。

preempt：接收一个 Controller 事件，并抢占队列之前的事件进行优先处理。

目前，在 Kafka 源码中，KafkaController 类是 Controller 组件的功能实现类，它也是 ControllerEventProcessor 接口的唯一实现类。

对于这个接口，你要重点掌握 process 方法的作用，因为**它是实现 Controller 事件处理的主力方法**。你要了解 process 方法**处理各类 Controller 事件的代码结构是什么样的**，而且还要能够准确地找到处理每类事件的子方法。

至于 preempt 方法，你仅需要了解，Kafka 使用它实现某些高优先级事件的抢占处理即可，毕竟，目前在源码中只有两类事件（ShutdownEventThread 和 Expire）需要抢占式处理，出镜率不是很高。

ControllerEvent

这就是前面说到的 Controller 事件，在源码中对应的就是 ControllerEvent 接口。该接口定义在 KafkaController.scala 文件中，本质上是一个 trait 类型，如下所示：

复制代码

```
sealed trait ControllerEvent {
  def state: ControllerState
}
```

每个 ControllerEvent 都定义了一个状态。Controller 在处理具体的事件时，会对状态进行相应的变更。这个状态是由源码文件 ControllerState.scala 中的抽象类 ControllerState 定义的，代码如下：

复制代码

```
sealed abstract class ControllerState {
  def value: Byte
  def rateAndTimeMetricName: Option[String] =
    if (hasRateAndTimeMetric) Some(s"${toString}RateAndTimeMs") else None
  protected def hasRateAndTimeMetric: Boolean = true
}
```

每类 ControllerState 都定义一个 value 值，表示 Controller 状态的序号，从 0 开始。另外，rateAndTimeMetricName 方法是用于构造 Controller 状态速率的监控指标名称的。

比如，TopicChange 是一类 ControllerState，用于表示主题总数发生了变化。为了监控这类状态变更速率，代码中的 rateAndTimeMetricName 方法会定义一个名为

TopicChangeRateAndTimeMs 的指标。当然，并非所有的 ControllerState 都有对应的速率监控指标，比如，表示空闲状态的 Idle 就没有对应的指标。

目前，Controller 总共定义了 25 类事件和 17 种状态，它们的对应关系如下表所示：

<table>
<tr><th>ControllerEvent</th><th>ControllerState</th></tr>
<tr><td>ControllerChange</td><td rowspan="5">ControllerChange</td></tr>
<tr><td>Reelect</td></tr>
<tr><td>RegisterBrokerAndReelect</td></tr>
<tr><td>Expire</td></tr>
<tr><td>Startup</td></tr>
<tr><td>ShutdownEventThread</td><td>ControllerShutdown</td></tr>
<tr><td>AutoPreferredReplicaLeaderElection</td><td>AutoLeaderBalance</td></tr>
<tr><td>UncleanLeaderElectionEnable</td><td>UncleanLeaderElectionEnable</td></tr>
<tr><td>TopicUncleanLeaderElectionEnable</td><td>TopicUncleanLeaderElectionEnable</td></tr>
<tr><td>ControlledShutdown</td><td>ControlledShutdown</td></tr>
<tr><td>LeaderAndIsrResponseReceived</td><td>LeaderAndIsrResponseReceived</td></tr>
<tr><td>UpdateMetadataResponseReceived</td><td>UpdateMetadataResponseReceived</td></tr>
<tr><td>TopicDeletionStopReplicaResponseReceived</td><td rowspan="2">TopicDeletion</td></tr>
<tr><td>TopicDeletion</td></tr>
<tr><td>BrokerChange</td><td rowspan="2">BrokerChange</td></tr>
<tr><td>BrokerModifications</td></tr>
<tr><td>TopicChange</td><td rowspan="2">TopicChange</td></tr>
<tr><td>PartitionModifications</td></tr>
<tr><td>LogDirEventNotification</td><td>LogDirChange</td></tr>
<tr><td>ZkPartitionReassignment</td><td></td></tr>
</table>

ApiPartitionReassignment	AlterPartitionReassignment
PartitionReassignmentIsrChange	
IsrChangeNotification	IsrChange
ReplicaLeaderElection	ManualLeaderBalance
ListPartitionReassignments	ListPartitionReassignment

极客时间

内容看着好像有很多，那我们应该怎样使用这张表格呢？

实际上，你并不需要记住每一行的对应关系。这张表格更像是一个工具，当你监控到某些 Controller 状态变更速率异常的时候，你可以通过这张表格，快速确定可能造成瓶颈的 Controller 事件，并定位处理该事件的函数代码，辅助你进一步地调试问题。

另外，你要了解的是，**多个 ControllerEvent 可能归属于相同的 ControllerState。**

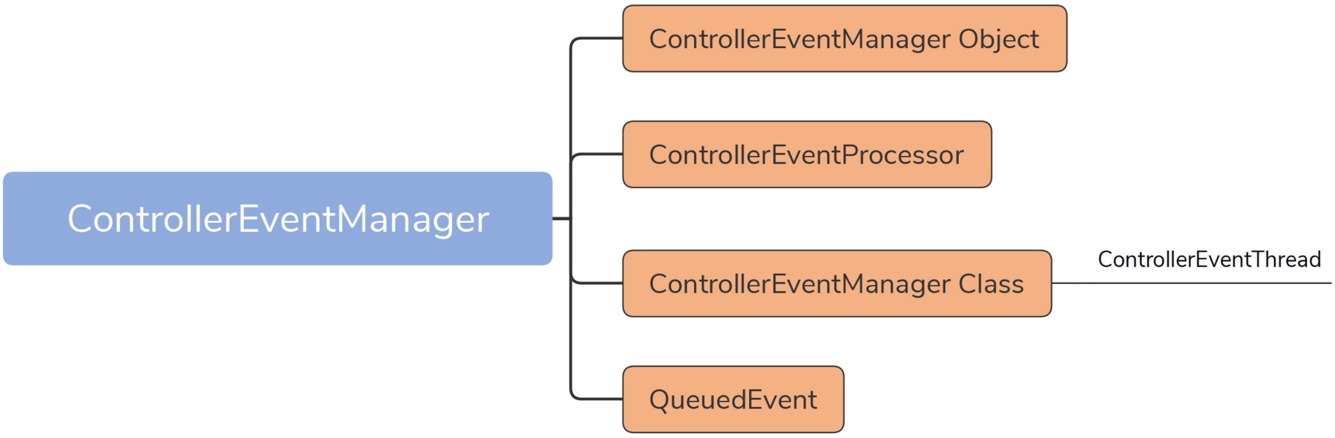
比如，TopicChange 和 PartitionModifications 事件都属于 TopicChange 状态，毕竟，它们都与 Topic 的变更有关。前者是创建 Topic，后者是修改 Topic 的属性，比如，分区数或副本因子，等等。

再比如，BrokerChange 和 BrokerModifications 事件都属于 BrokerChange 状态，表征的都是对 Broker 属性的修改。

ControllerEventManager

有了这些铺垫，我们就可以开始学习事件处理器的实现代码了。

在 Kafka 中，Controller 事件处理器代码位于 controller 包下的 ControllerEventManager.scala 文件下。我用一张图来展示下这个文件的结构：



极客时间

如图所示，该文件主要由 4 个部分组成。

- **ControllerEventManager Object**：保存一些字符串常量，比如线程名字。
- **ControllerEventProcessor**：前面讲过的事件处理器接口，目前只有 KafkaController 实现了这个接口。
- **QueuedEvent**：表征事件队列上的事件对象。
- **ControllerEventManager Class**：ControllerEventManager 的伴生类，主要用于创建和管理事件处理线程和事件队列。就像我前面说的，这个类中定义了重要的 ControllerEventThread 线程类，还有一些其他值得我们学习的重要方法，一会儿我们详细说说。

ControllerEventManager 对象仅仅定义了 3 个公共变量，没有任何逻辑，你简单看下就行。至于 ControllerEventProcessor 接口，我们刚刚已经学习过了。接下来，我们重点学习后面这两个类。

QueuedEvent

我们先来看 QueuedEvent 的定义，全部代码如下：

复制代码

```
// 每个QueuedEvent定义了两个字段
// event: ControllerEvent类，表示Controller事件
// enqueueTimeMs: 表示Controller事件被放入到事件队列的时间戳
class QueuedEvent(val event: ControllerEvent,
```

```
                        val enqueueTimeMs: Long) {
  // 标识事件是否开始被处理
  val processingStarted = new CountDownLatch(1)
  // 标识事件是否被处理过
  val spent = new AtomicBoolean(false)
  // 处理事件
  def process(processor: ControllerEventProcessor): Unit = {
    if (spent.getAndSet(true))
      return
    processingStarted.countDown()
    processor.process(event)
  }
  // 抢占式处理事件
  def preempt(processor: ControllerEventProcessor): Unit = {
    if (spent.getAndSet(true))
      return
    processor.preempt(event)
  }
  // 阻塞等待事件被处理完成
  def awaitProcessing(): Unit = {
    processingStarted.await()
  }
  override def toString: String = {
    s"QueuedEvent(event=$event, enqueueTimeMs=$enqueueTimeMs)"
  }

```

可以看到，每个 QueuedEvent 对象实例都裹挟了一个 ControllerEvent。另外，每个 QueuedEvent 还定义了 process、preempt 和 awaitProcessing 方法，分别表示**处理事件**、**以抢占方式处理事件**，以及**等待事件处理**。

其中，process 方法和 preempt 方法的实现原理，就是调用给定 ControllerEventProcessor 接口的 process 和 preempt 方法，非常简单。

在 QueuedEvent 对象中，我们再一次看到了 CountDownLatch 的身影，我在第 7 节课里提到过它。Kafka 源码非常喜欢用 CountDownLatch 来做各种条件控制，比如用于侦测线程是否成功启动、成功关闭，等等。

在这里，QueuedEvent 使用它的唯一目的，是确保 Expire 事件在建立 ZooKeeper 会话前被处理。

如果不是在这个场景下，那么，代码就用 spent 来标识该事件是否已经被处理过了，如果已经被处理过了，再次调用 process 方法时就会直接返回，什么都不做。

ControllerEventThread

了解了 QueuedEvent，我们来看下消费它们的 ControllerEventThread 类。

首先是这个类的定义代码：

复制代码

```
class ControllerEventThread(name: String) extends ShutdownableThread(name = na
  logIdent = s"[ControllerEventThread controllerId=$controllerId] "
  ......
}
```

这个类就是一个普通的线程类，继承了 ShutdownableThread 基类，而后者是 Kafka 为很多线程类定义的公共父类。该父类是 Java Thread 类的子类，其线程逻辑方法 run 的主要代码如下：

复制代码

```
def doWork(): Unit
override def run(): Unit = {
  ......
  try {
    while (isRunning)
      doWork()
  } catch {
    ......
  }
  ......
}
```

可见，这个父类会循环地执行 doWork 方法的逻辑，而该方法的具体实现则交由子类来完成。

作为 Controller 唯一的事件处理线程，我们要时刻关注这个线程的运行状态。因此，我们必须要知道这个线程在 JVM 上的名字，这样后续我们就能有针对性地对其展开监控。这个线程的名字是由 ControllerEventManager Object 中 ControllerEventThreadName 变量定义的，如下所示：

复制代码

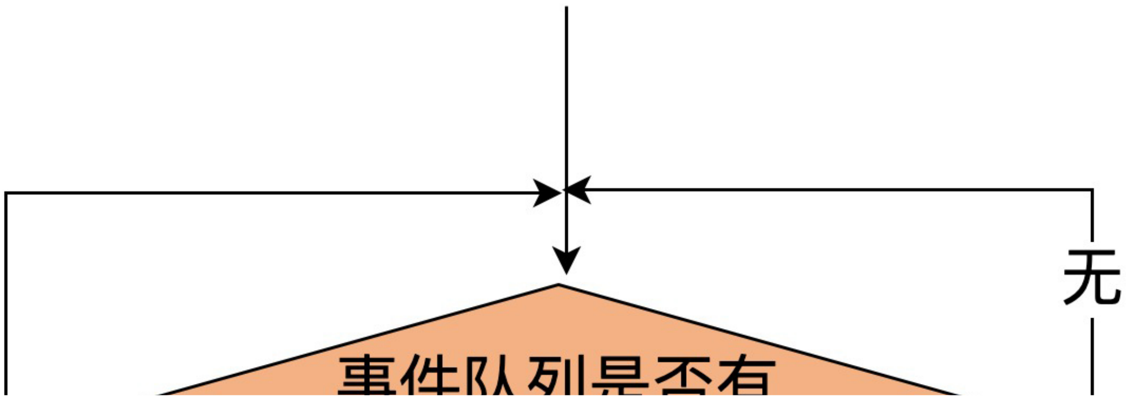
```
object ControllerEventManager {
  val ControllerEventThreadName = "controller-event-thread"
  ......
```

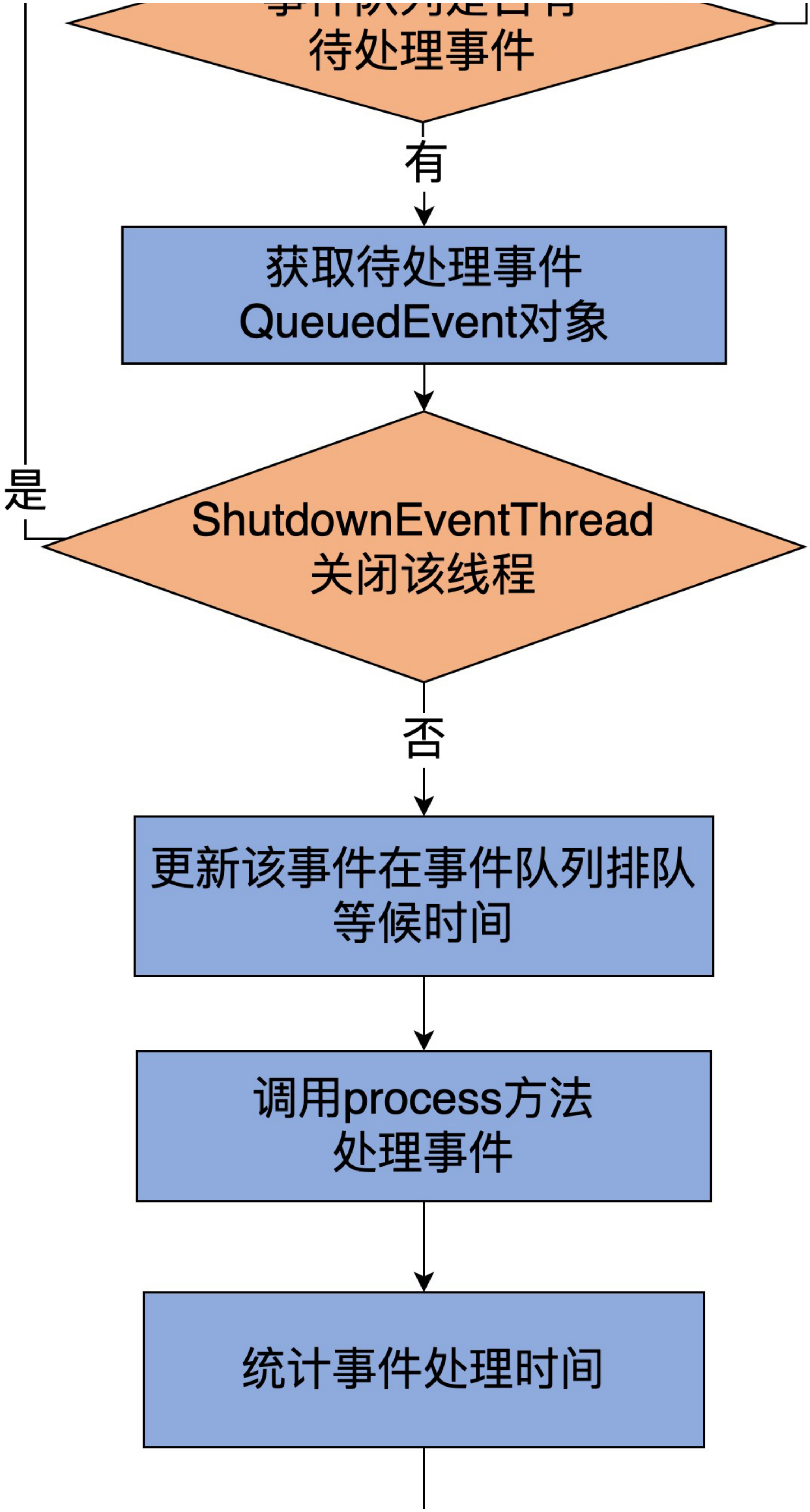
现在我们看看 ControllerEventThread 类的 doWork 是如何实现的。代码如下：

复制代码

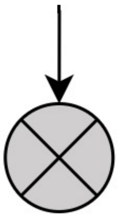
```
override def doWork(): Unit = {
  // 从事件队列中获取待处理的Controller事件，否则等待
  val dequeued = queue.take()
  dequeued.event match {
    // 如果是关闭线程事件，什么都不用做。关闭线程由外部来执行
    case ShutdownEventThread =>
    case controllerEvent =>
      _state = controllerEvent.state
      // 更新对应事件在队列中保存的时间
      eventQueueTimeHist.update(time.milliseconds() - dequeued.enqueueTimeMs)
      try {
        def process(): Unit = dequeued.process(processor)
        // 处理事件，同时计算处理速率
        rateAndTimeMetrics.get(state) match {
          case Some(timer) => timer.time { process() }
          case None => process()
        }
      } catch {
        case e: Throwable => error(s"Uncaught error processing event $controll
      }
      _state = ControllerState.Idle
  }
}
```

我用一张图来展示下具体的执行流程：





待处理事件
有
获取待处理事件
QueuedEvent对象
是
ShutdownEventThread
关闭该线程
否
更新该事件在事件队列排队
等候时间
调用process方法
处理事件
统计事件处理时间



大体上看，执行逻辑很简单。

首先是调用 LinkedBlockingQueue 的 take 方法，去获取待处理的 QueuedEvent 对象实例。注意，这里用的是 **take 方法**，这说明，如果事件队列中没有 QueuedEvent，那么，ControllerEventThread 线程将一直处于阻塞状态，直到事件队列上插入了新的待处理事件。

一旦拿到 QueuedEvent 事件后，线程会判断是否是 ShutdownEventThread 事件。当 ControllerEventManager 关闭时，会显式地向事件队列中塞入 ShutdownEventThread，表明要关闭 ControllerEventThread 线程。如果是该事件，那么 ControllerEventThread 什么都不用做，毕竟要关闭这个线程了。相反地，如果是其他的事件，就调用 QueuedEvent 的 process 方法执行对应的处理逻辑，同时计算事件被处理的速率。

该 process 方法底层调用的是 ControllerEventProcessor 的 process 方法，如下所示：

复制代码

```
def process(processor: ControllerEventProcessor): Unit = {
  // 若已经被处理过，直接返回
  if (spent.getAndSet(true))
    return
  processingStarted.countDown()
  // 调用ControllerEventProcessor的process方法处理事件
  processor.process(event)
}
```

方法首先会判断该事件是否已经被处理过，如果是，就直接返回；如果不是，就调用 ControllerEventProcessor 的 process 方法处理事件。

你可能很关心，每个 ControllerEventProcessor 的 process 方法是在哪里实现的？实际上，它们都封装在 KafkaController.scala 文件中。还记得我之前说过，KafkaController 类是目前源码中 ControllerEventProcessor 接口的唯一实现类吗？

实际上，就是 KafkaController 类实现了 ControllerEventProcessor 的 process 方法。由于代码过长，而且有很多重复结构的代码，因此，我只展示部分代码：

复制代码

```
override def process(event: ControllerEvent): Unit = {
    try {
      // 依次匹配ControllerEvent事件
      event match {
        case event: MockEvent =>
          event.process()
        case ShutdownEventThread =>
          error("Received a ShutdownEventThread event. This type of event is s
        case AutoPreferredReplicaLeaderElection =>
          processAutoPreferredReplicaLeaderElection()
        ......
      }
    } catch {
      // 如果Controller换成了别的Broker
      case e: ControllerMovedException =>
        info(s"Controller moved to another broker when processing $event.", e)
        // 执行Controller卸任逻辑
        maybeResign()
      case e: Throwable =>
        error(s"Error processing event $event", e)
    } finally {
      updateMetrics()
    }
}
```

这个 process 方法接收一个 ControllerEvent 实例，接着会判断它是哪类 Controller 事件，并调用相应的处理方法。比如，如果是 AutoPreferredReplicaLeaderElection 事件，则调用 processAutoPreferredReplicaLeaderElection 方法；如果是其他类型的事件，则调用 process*** 方法。

其他方法

除了 QueuedEvent 和 ControllerEventThread 之外，**put 方法**和 **clearAndPut 方法也很重要**。如果说 ControllerEventThread 是读取队列事件的，那么，这两个方法就是向队

列生产元素的。

在这两个方法中，put 是把指定 ControllerEvent 插入到事件队列，而 clearAndPut 则是先执行具有高优先级的抢占式事件，之后清空队列所有事件，最后再插入指定的事件。

下面这两段源码分别对应于这两个方法：

复制代码

```
// put方法
def put(event: ControllerEvent): QueuedEvent = inLock(putLock) {
  // 构建QueuedEvent实例
  val queuedEvent = new QueuedEvent(event, time.milliseconds())
  // 插入到事件队列
  queue.put(queuedEvent)
  // 返回新建QueuedEvent实例
  queuedEvent
}
// clearAndPut方法
def clearAndPut(event: ControllerEvent): QueuedEvent = inLock(putLock) {
  // 优先处理抢占式事件
  queue.forEach(_.preempt(processor))
  // 清空事件队列
  queue.clear()
  // 调用上面的put方法将给定事件插入到事件队列
  put(event)
}
```

整体上代码很简单，需要解释的地方不多，但我想和你讨论一个问题。

你注意到，源码中的 put 方法使用 putLock 对代码进行保护了吗？

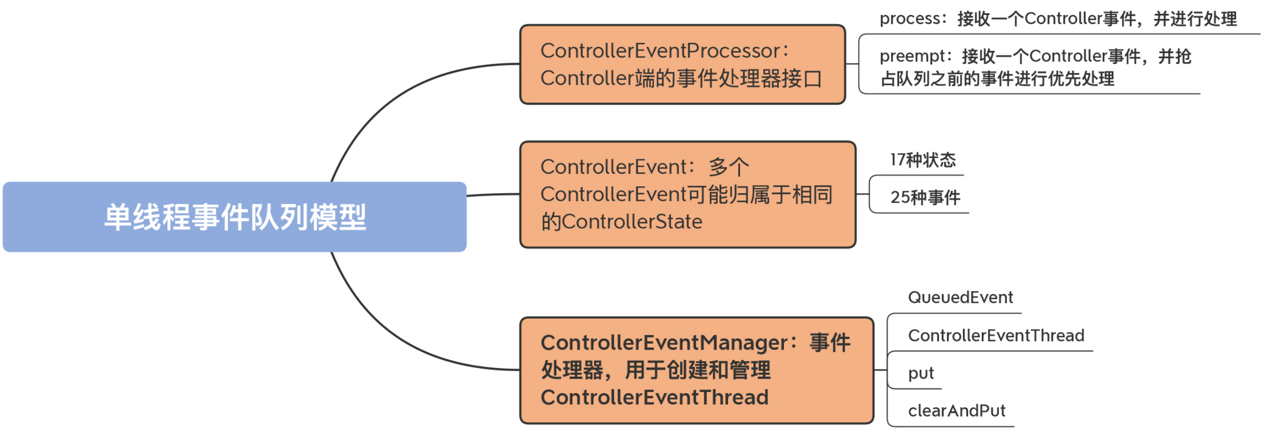
就我个人而言，我觉得这个 putLock 是不需要的，因为 LinkedBlockingQueue 数据结构本身就已经是线程安全的了。put 方法只会与全局共享变量 queue 打交道，因此，它们的线程安全性完全可以委托 LinkedBlockingQueue 实现。更何况，LinkedBlockingQueue 内部已经维护了一个 putLock 和一个 takeLock，专门保护读写操作。

当然，我同意在 clearAndPut 中使用锁的做法，毕竟，我们要保证，访问抢占式事件和清空操作构成一个原子操作。

总结

今天，我们重点学习了 Controller 端的单线程事件队列实现方式，即 ControllerEventManager 通过构建 ControllerEvent、ControllerState 和对应的 ControllerEventThread 线程，并且结合专属事件队列，共同实现事件处理。我们来回顾下这节课的重点。

- ControllerEvent：定义 Controller 能够处理的各类事件名称，目前总共定义了 25 类事件。
- ControllerState：定义 Controller 状态。你可以认为，它是 ControllerEvent 的上一级分类，因此，ControllerEvent 和 ControllerState 是多对一的关系。
- ControllerEventManager：Controller 定义的事件管理器，专门定义和维护专属线程以及对应的事件队列。
- ControllerEventThread：事件管理器创建的事件处理线程。该线程排他性地读取事件队列并处理队列中的所有事件。



极客时间

下节课，我们将正式进入到 KafkaController 的学习。这是一个有着 2100 多行的大文件，不过大部分的代码都是实现那 27 类 ControllerEvent 的处理逻辑，因此，你不要被它吓到了。我们会先学习 Controller 是如何选举出来的，后面会再详谈 Controller 的具体作用。

课后讨论

你认为，ControllerEventManager 中 put 方法代码是否有必要被一个 Lock 保护起来？

欢迎你在留言区畅所欲言，跟我交流讨论，也欢迎你把今天的内容分享给你的朋友。

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 12 | ControllerChannelManager：Controller如何管理请求发送?

下一篇 14 | Controller选举是怎么实现的?

精选留言 (4)

写留言

胡夕 置顶
2020-05-19

你好，我是胡夕。我来公布上节课的“课后讨论”题答案啦～

上节课，咱们重点了解了Controller通道管理器类ControllerChannelManager。课后我请你思考这样一个问题，即为每个Broker都创建一个RequestSendThread的方案有什么优缺点。Controller为每个Broker都创建专属的线程，可以隔离与其他Broker之间的交互，...

展开

2

伯安知心
2020-05-20

本人最近找大数据开发运维之类的工作，有没有推荐呢?各位亲

RonnieXie

2020-05-20

目前我学习的源码，理解到ControllerEventThread也是只有一个进程处理，
是否实现ControllerEventThread每个Broker都创建专属的线程，就可以同时优化Request SendThread和ControllerEventThread单经常处理的缺点?

展开

作者回复: 嗯，也许是个不错的方案。你可以尝试实现下，可能是个很好的功能改进

伯安知心

2020-05-20

从put代码看，只有2步骤，增加时间戳封装队列事件，然后加入linked有序阻塞队列，也许kafka是为了保持加入顺序吧，但是这个阻塞队列本身就是线程安全的，确实有些鸡肋，个人观点。

作者回复: 有一定道理；0

1